

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Patent Application

Applicants(s): Bohannon et al
Case: 15-1-14-2
Serial No : 10/766,305
Filing Date: January 28, 2004
Group: 2176
Examiner: C Adams
Title: Method and Apparatus for Updating XML Views of Relational Data

AFFIDAVIT UNDER 37 C.F.R. §1.131

We, the undersigned, hereby declare and state as follows:

1. We are the named inventors on the above-referenced U.S. patent application
2. We conceived the invention that is the subject matter of one or more claims of the above-referenced application at least as early as November 14, 2002. On or about November 14, 2002, we prepared a Disclosure of Invention entitled Updating XML Views of Relational Data. A copy of the Disclosure of Invention is attached hereto as Exhibit 1
3. A related paper entitled Updating XML Views of Relational Data (Paper Number 332) was attached to the Disclosure of Invention form. The paper was also submitted for publication in ACM SIGMOD 2003. A copy of the paper is attached hereto as Exhibit 2.
4. The invention was reduced to practice by implementing it in software code prior to or in conjunction with the preparation of the paper. The software code

embodying the invention was used to obtain the initial experimental results referred to in Section 4.4 (entitled Implementation) of the cited paper. The cited test results indicating correct operation are evidence of an actual reduction to practice of one embodiment of the present invention

5 The implemented algorithm contained an Information Collection Module and a View-Update Execution Module. See, Par 4 4

Information-Collection Module

6 The Information-Collection Module collects the static information described in Section 4.2 of the paper. The view-relationship graph is then translated into update plans that are persisted in the system and later used at run time. *Id*

7 Claim 1 generally requires assigning at least one of a plurality of categories to each of the nodes. The plurality of categories are based on a cardinality relationship indicated by one or more correlation predicates and one or more foreign key constraints in the relational database.

8. Among other things, the implementation collects the category of the node. See, Section 4.2. This is obtained from the view query that defines the XML view and the algorithms defined in Section 3. Algorithm 1 in section 3 describes how the node is categorized.

9 As stated in section 3.2, the implemented node categorization is based upon cardinality relationships indicated by foreign-key constraints in the underlying relational database.

10. The implementation of the Information-Collection Module was a reduction to practice of a software program that assigned at least one of a plurality of

categories to each of the nodes, where the plurality of categories are based on a cardinality relationship indicated by one or more correlation predicates and one or more foreign key constraints in the relational database.

View-Update Execution Module

11. The View-Update Execution Module provides the interface for deletion, insertion, movement, and replacement (deleting the old node and inserting the new node in one transaction) on a given XML node at run time. The execution module interacts with the relational database and the DOM interface to access the underlying data for the XML view. See, Par. 4.4.

12. Claim 1 generally requires determining whether the update to the XML document can be updated in the underlying relational database based on the assigned category.

13. The implementation of Algorithm 2, discussed in section 3.3, provides an example of how to translate the deletion of a node from an XML view document into base view updates, and the categories for which such deletions cannot be performed. Generally, Algorithm 2 shows how a deletion is performed for each category.

14. The implementation of the View-Update Execution Module was a reduction to practice of a software program that determined whether an update to the XML document can be updated in the underlying relational database based on an assigned category.

15. Claims 12 and 16 have similar limitations to claim 1.

16. The work noted in this Affidavit took place at the Murray Hill Facility of Lucent Technologies in Murray Hill, New Jersey.

of how to translate the deletion of a node from an XML view document into base view updates, and the categories for which such deletions cannot be performed. Generally, Algorithm 2 shows how a deletion is performed for each category

14. The implementation of the View-Update Execution Module was a reduction to practice of a software program that determined whether an update to the XML document can be updated in the underlying relational database based on an assigned category.

15. Claims 12 and 16 have similar limitations to claim 1.

16. The work noted in this Affidavit took place at the Murray Hill Facility of Lucent Technologies in Murray Hill, New Jersey.

17. All statements made herein of our own knowledge are true, and all statements made on information and belief are believed to be true.

18. I understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: 12/18/2006



Philip Bohannon

Date: _____

Xin Dong

Date: _____

Henry F. Korth

Date: _____

Suryanarayan Perinkulam

Bohannon 15-1-14-2

17 All statements made herein of our own knowledge are true, and all statements made on information and belief are believed to be true

18 I understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U S C. §1001, and may jeopardize the validity of the application or any patent issuing thereon

Date: _____

Date: 12/18/06

Date: _____

Date: _____

Philip Bohannon

Xin Dong

Henry F. Korth

Suryanarayan Perinkulam

of how to translate the deletion of a node from an XML view document into base view updates, and the categories for which such deletions cannot be performed. Generally, Algorithm 2 shows how a deletion is performed for each category.

14. The implementation of the View-Update Execution Module was a reduction to practice of a software program that determined whether an update to the XML document can be updated in the underlying relational database based on an assigned category.

15. Claims 12 and 16 have similar limitations to claim 1.

16. The work noted in this Affidavit took place at the Murray Hill Facility of Lucent Technologies in Murray Hill, New Jersey.

17. All statements made herein of our own knowledge are true, and all statements made on information and belief are believed to be true.

18. I understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____

Philip Bohannon

Date: _____

Xin Dong

Date: 12-18-2006



Henry F. Korth

Date: _____

Suryanarayan Perinkulam

17 All statements made herein of our own knowledge are true, and all statements made on information and belief are believed to be true.

18 I understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____

Philip Bohannon

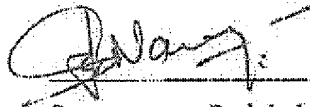
Date: _____

Xin Dong

Date: _____

Henry F. Korth

Date: Nov Dec 2006


Suryanarayan Perinkulam

Exh 1

Class number: <u>3</u>	Date <u>11/15/02</u>
AS	

For IP-Law Use Only

LUCENT TECHNOLOGIES INC.

1-908-582-5809

PAUL BOHANNON

3 SHEETS

DISCLOSURE OF INVENTION

THIS DISCLOSURE SHOULD BE SUPPLEMENTED BY ATTACHING COPIES OF RELEVANT DOCUMENTS SUCH AS TECHNICAL MEMORANDA, PUBLISHED OR TO-BE-PUBLISHED ARTICLES AND ENGINEERING NOTEBOOK PAGES.

(Also, if for any item there is insufficient space on the form, attach additional pages as necessary.)

DESCRIPTIVE TITLE OF INVENTION: Updating XML Views of Relational Data

INVENTOR #1: <u>Xing Dong</u>	<u>University of Washington</u>
<u>lunadong@cs.washington.edu;</u>	<u>Company/Location</u>
<u>Phone/E-mail</u>	<u>Director's Name</u>

INVENTOR #2: <u>P. Bohannon</u>	<u>Lucent Technologies</u>
<u>908-582-7695 bohannon@research.bell-labs.com</u>	<u>Company/Location</u>
<u>Phone/E-mail</u>	<u>Director's Name</u>

INVENTOR #3: <u>H.F. Korth</u>	<u>Lucent Technologies</u>
<u>908-582-7791 hfk@research.bell-labs.com</u>	<u>Company/Location</u>
<u>Phone/E-mail</u>	<u>Director's Name</u>

INVENTOR #4 <u>P.P. S. Narayan</u>	<u>Lucent Technologies</u>
<u>908-582-4314 ppsn@research.bell-labs.com</u>	<u>Company/Location</u>
<u>Phone/E-mail</u>	<u>Director's Name</u>

1 PRIMARY CONTACT

If more than one inventor is named above, who will have the primary responsibility for interfacing with Lucent IP-Law with respect to preparing and prosecuting a patent application for the invention?

Inventor Name: H.F. Korth

2 PRESENT STATE OF THE INVENTION

☐ Idea ☒ Research ☐ Development

☐ Manufacture (Product Name _____ Ship Date _____)

Lucent Technologies Inc - Proprietary
Use Pursuant To Company Instructions

3. GOVERNMENT CONTRACT INVENTION

Was the invention made under a government contract? ☐ Yes ☒ No

4. PRESENT STATE OF THE ART

Briefly describe the closest already-known technology that relates to the invention. This would include, for example, already existing products, methods or compositions which are known to you personally or through descriptions in publications or patents

There are three practical approaches addressing the view-update problem in a general setting (typically relational, and not addressing any XML-specific issues). One is to regard the underlying database and the view as abstract data types, with the updating operations predefined explicitly by the DBA [12, 13]. The second determines a unique or a small set of update translations based on the syntax and semantics of a view definition [7, 10]. The algorithm presented in [10], given that the underlying base tables are in Boyce-Codd-Normal-Form, generates a query graph for the select-project-join view, and, based on the graph, gives a list of templates for possible translations of deletion, insertion and replacement operations on the view into certain update operations on the underlying database. This work is further extended by [4] for object-based views. The third approach performs run-time translation [14] transforms the view-update problem into the constraint-satisfaction problem (CSP), with the exponential time complexity in the number of constraint variables. [6] gives run-time translations of view tuple deletions using data lineage, but claims the current state of the art in data lineage is not applicable to other view update operations like insertions. Recent work on XML updates [17] studies this problem in the context of XML shredding using the inlining method [16]. Inlining defines a specific procedure for the conversion of a *given* XML schema into a relational schema and the storage of XML data in a relational database conforming to that schema. The original XML schema alone determines the update strategy. Thus, the prior work on updates specifically applying to XML-defined views considers only a case where the underlying database fits a particular type of database schema. The work here considers any well defined schema.

5. ADVANCEMENT IN STATE OF THE ART

Briefly describe the unique advancement achieved by the invention. This may be done, for example, by describing a problem with the prior art that is solved or specific objects that are achieved by the invention.

There are three characteristics of our invention that are unique:

- 1) The relational database schema and constraints on which the XML views are defined are predefined and arbitrary. That is, our algorithm works based on any arbitrary given schema that is predefined by a pre-existing database design.
- 2) The XML view is defined based on the database, whereas prior work on XML views assumes the database is defined based on a given set of XML data.
- 3) The XML view and updates through that view are always synchronized with the base (relational) data, whereas current systems treat XML views as a non-updateable cache.

6. HOW ACHIEVED

Briefly describe the invention and how it achieves the advancement described in paragraph 6

In this paper, we develop a framework for XML updates that is sufficiently general and flexible to deal with *any* XML view over virtually *any* relational schema. Based on this, we present algorithms to translate an XML update into update operations on the underlying relational database. These algorithms are based on a "view-relationship graph" that encapsulates relationships and constraints on view data based upon both the view definition and the underlying relational database schema. We then partition the graph based on the graph structure such that each partition has specific well-defined properties in terms of how updates are to be performed (or disallowed). In addition, we describe an implementation

on top of an existing data system, the ROLEX system that will run on top of the DataBlitz™ main memory database system (these two systems are not named in the paper as part of our effort to anonymize our submission). Our framework ensures the consistency of the updated database, and takes into consideration constraints both in the underlying relational database and in the XML-view definition

7 DISCLOSURE OUTSIDE OF LUCENT

Anticipated Publications Date: 6/9-12/03 Publication Name: ACM SIGMOD 2003

Submitted to Publication Clearance? ☒ Yes ☐ No

If the invention was or will otherwise be disclosed to any non-Lucent employee, describe to whom (person/company), when, where, why, and whether it was/will be under a non-disclosure agreement

Dong has continued to work on this paper after her employment at Lucent ended. The bulk of her work was done during a summer internship at Bell Labs.

8. Have any of the submitters discussed this invention with an attorney (either Lucent or outside) other than Steve Gurey? Yes _____ No x

If Yes, give name of the attorney: _____

9. Is this invention related to another Lucent invention submission, patent application, or issued patent? Yes _____ No x (Please check Yes or No only. If Yes, attorney will contact you for further information)

10 INVENTOR #1 _____

Signature

Date

INVENTOR #2

Signature

Date

INVENTOR #3

Signature

Date

INVENTOR #4

Signature

Date

Updating XML Views of Relational Data

Paper number 332

ABSTRACT

Recent XML middleware systems bridge between XML applications and relational database systems by supporting XML publishing and querying. Update operations, however, are not well supported, particularly when the underlying relational database not only serves XML applications, but also is accessed directly by relational applications. In this paper, we develop a framework for XML updates that is sufficiently general and flexible to deal with *any* XML view over virtually *any* relational schema. Based on this, we present algorithms to translate an XML update into update operations on the underlying relational database. In addition, we describe our implementation on top of an existing database system. Our framework ensures the consistency of the updated database, and takes into consideration constraints both in the underlying relational database and in the XML-view definition. The experimental results show that the system operates correctly with reasonable performance.

1. INTRODUCTION

In recent years, XML has gained widespread popularity and has begun to play an important role for information representation and exchange. Although XML-based applications may be developed from scratch, in most cases they interoperate with existing SQL-centric applications. Recent work addresses the techniques required to define the mapping between XML data and relations, to format SQL output as XML, and to translate queries posed in XML query languages into SQL [1, 9, 15]. The next step in making them into full-featured interoperation tools is to explore the management of update operations.

We consider the case of an XML-based application that is built upon an existing relational database that serves traditional RDBMS applications as well. There are three characteristics of this architecture: (1) the relational database schema and constraints are predefined and arbitrary; (2) the XML view is defined based on the database; and, (3) the XML view and updates through that view are always synchronized with the base (relational) data. This contrasts with prior work (which we discuss in more detail below) in which XML views are read-only or in which the underlying relational schema is restricted to be one derivable via "XML shredding." Our work on updates through XML views is closely related to the vast body of earlier work on updates to relational databases through relational views.

Many of the problems addressed in that work apply to our domain as well, and this paper focuses on those aspects unique to our assumption of XML-defined views.

We base our work on the *redacted*¹ system, which provides a declarative language to extract data from an *existing* relational database and generate an XML version of data. This system provides all of the basic features of standard commercial relational products. Regardless of the specific underlying relational system, our approach of supporting updates through XML views allows such issues as concurrency, recovery, and many aspects of consistency and integrity checking to be done by the underlying database system. This contrasts to XML-publishing environments such as [15] or commercial relational systems, in which each application typically caches its own materialized XML view and much of the capability of the underlying database system goes unused.

1.1 Previous Work Based on Shredding

The recent work on XML updates [17] studies this problem in the context of XML shredding using the inlining method [16]. Inlining defines a specific procedure for the conversion of a *given* XML schema into a relational schema and the storage of XML data in a relational database conforming to that schema. The original XML schema alone determines the update strategy.

In our framework, we regard the relational schema as predefined and essentially arbitrary. Both the relational schema and the XML schema form the input of the update problem in our work. Consequently, many more possible cases need to be considered. The solution must be general and flexible enough to deal with *any* XML view over *any* relational schema. In particular, our approach is suitable for existing relational databases that are now to be accessed by XML-based applications as well.

Two examples can demonstrate the complexity in handling the side-effects of updates and contrast the assumptions of [17] with those of our approach:

Example 1: Given the relational database and its schema as shown in Figure 1(a), and the view definition² shown in Figure 1(b), the resulting XML document fragment is shown in Figure 1(c). Using inlining algorithm as in [17], a different set of base tables with a different relational schema would be generated from the view, as demonstrated in Figure 1(d). In the latter schema, the *Metroarea* relation, rather than the *Hotel* relation, has a foreign key. Instead of having a single tuple for each metro area, there is a separate *Metroarea* tuple for each hotel. When a hotel node in the document is deleted, it is easy to see that all

¹ The name of the system has been redacted (removed) to conform to SIGMOD's double-blind review policy. Details will appear in the final version of the paper.

² The *tag-query* notation is derived from that of SilkRoute [9].

Metroarea (mID, mName)
Hotel (hID hName m_id)

mID	mName
101	Northwest

hID	hName	m_id
201	Courtyard	101
202	Doubletree	101

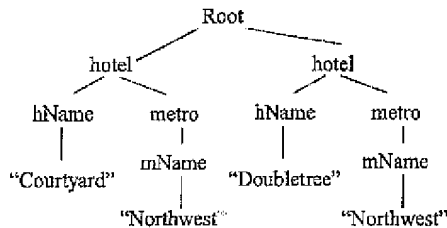
(a) Original schema and database

```

<hotel>
($h = SELECT hName
FROM Hotel
)
<metro> ($m = SELECT mName
FROM Metroarea
WHERE mID = $h.m_id
)</metro>
</hotel>

```

(b) View definition



(c) XML document fragment

Hotel (hID hName)
Metroarea (mID mName, h_id)

mID	mName	h_id
101	Northwest	201
102	Northwest	202

hID	hName
201	Courtyard
202	Doubletree

(d) Inlining generated schema and database

Figure 1: Example 1

the tuples related to its child nodes, more specifically, the related Metroarea tuples must be deleted. However, in the original database of Figure 1(a), the Metroarea tuple must be preserved for other hotels. As we (unlike [17]) permit either database schema as our underlying schema, we need to find a way to distinguish them and give different update plans.

Example 2: Figure 2(a) shows a relational schema, and Figure 2(b) shows a view definition. The original relational schema is made up of three relations, while the inlining-generated schema based on the XML view shown in Figure 2(c) consists of only two relations. Given the latter schema, the deletion of a metro node can be executed in a straightforward manner by propagating the deletion to the Confroom table. If we decide to do the same thing under the original relational schema, the Hotel table is affected due to the foreign-key constraint. As the Hotel table is invisible to the XML application, we consider any operation on it undesirable.

Metroarea (mID, mName)
Hotel (hID hName, m_id)
Confroom (cID roomnum, h_id)

(a) Original schema

```

<metro>
($m = SELECT mName
FROM Metroarea)
<conference-room>
($c = SELECT cID, roomnum m_id
FROM Confroom, Hotel
WHERE Confroom.h_id = Hotel hID
AND Hotel m_id = $m mID
)</conference-room>
</metro>

```

(b) View definition

Metroarea (mID, mName)
Confroom (cID roomnum m_id)

(c) Inlining generated schema

Figure 2: Example 2

The above examples show that updates through XML views are more challenging to manage when the underlying relational database schema is arbitrary and not the one derived from the view via inlining. The challenge arises from the fact that the XML view does not determine a unique relational database schema, and so, assumptions about the specific nature of the database schema cannot be built into the view-update algorithms.

1.2 Previous Work on View Updates

The view-update problem in relational databases is a long-standing issue that has been studied extensively. A survey of research on the view-update problem is presented in [8]. An abstract formulation of this problem is given by the view complementary theorem in [3]. [5] analyzes the complexity of automatically finding a minimal complement view for updates and shows the problem to be NP-complete. [2] proves that deciding whether a side-effect-free update solution exists is generally NP-hard.

There are three practical approaches addressing the view-update problem. One is to regard the underlying database and the view as abstract data types, with the updating operations predefined explicitly by the DBA [12, 13]. The second determines a unique or a small set of update translations based on the syntax and semantics of a view definition [7, 10]. The algorithm presented in [10], given that the underlying base tables are in Boyce-Codd-Normal-Form, generates a query graph for the select-project-join view, and, based on the graph, gives a list of templates for possible translations of deletion, insertion and replacement operations on the view into certain update operations on the underlying database. This work is further extended by [4] for object-based views. The third approach performs run-time translation [14] transforms the view-update problem into the constraint-satisfaction problem (CSP), with the exponential time complexity in the number of constraint variables. [6] gives run-time translations of view tuple deletions using data lineage, but claims the current state of the art in data lineage is not applicable to other view update operations like insertions.

Our XML view-update algorithm follows the line of the second approach. It shares the basic idea of deriving update methods from the view definition. We adapt the object concepts of [4] in our XML-based model. However, the XML model has features that distinguish it from the object model. For example, the XML document helps us decide the propagation direction. In Example 1, no matter whether the table *Hotel* has the foreign key pointing to *Metroarea* or the opposite, the propagation should follow the direction from *Hotel* to *Metroarea*. On the other hand, the possibility (indeed, the *reasonability*) of repeating certain data in different parts of an XML hierarchy raises more restrictive preconditions for an XML view to be updatable. Moreover, as an element can correspond to either a tuple or a field, the same type of update operation in XML needs to be translated into different kinds of relational update operations. Finally, some special XML view features such as transitive relationships and IDREF references (which we will discuss in Sections 3.4 and 3.5) bring more complication into the problem.

1.3 Contributions

In this paper, we describe the features of a working prototype of an XML view-update manager above the *redacted* system. Our system provides side-effect checking, DTD validation, constraint checking, and finally update translation and execution. The key contributions of this paper are summarized as follows:

- 1 We develop a framework for the determination of element updatability and the resulting impact on underlying tables. Based on this, we design algorithms for update translation.
- 2 After breaking up update operations into various sub-tasks such as DTD validation, constraint checking, and translation, we assign each sub-task to the XML view side or the relational database side as appropriate.
- 3 We present effective mechanisms to deal with constraints, re-organizing constraints from the database schema as well as the view schema, to improve performance.
- 4 Lastly, we implement the main part of the view update architecture in an existing system. Experiments show that it works correctly and that the performance is reasonable.

The paper is organized as follows. Section 2 discusses the XML view-update problem. Section 3 presents our view-update algorithms. Section 4 presents the architecture of our experimental system. We conclude and discuss future work in Section 5.

2. PROBLEM DEFINITION

In this section, we first list the set of allowed update operations and then introduce our running example.

2.1 XML Update Syntax and Semantics

In this paper, we do not focus on the specific syntax for the expression of XML updates. ([11], and others provide a syntax within XQuery for updates.) Regardless of the specific syntax used for XML updates, they can be divided into several categories as discussed below.

The first distinction we draw is among nodes of an XML document that are materialized from an XML view. A *text node*

in an XML document represents the string or numeral value of a PCDATA element or an attribute. We refer to this as a *value*. A *non-text node* (or *node*, when no confusion arises) is one that is not a value. We examine update operations only on nodes and not on values, as the latter can be transformed easily to the replacement of a node. Among the nodes in an XML document, the node representing a PCDATA element or an attribute is called a *leaf node*. Other non-root nodes are called *branch nodes*.

We consider XML update operations that touch the data but not the tags. Tag modification would result in schema change, which is beyond the scope of this paper. Data-update operations include deletion, insertion, movement, and replacement. In an XQuery-based syntax, these operations make use of the XQuery FLWR (FOR, LET, WHERE) statements: iterator, assignment, and conditional to locate the nodes for updates. We ignore order issues because the view-definition language used in *redacted* system does not offer a mechanism to define element order. These operations may be categorized as follows:

- A deletion is the removal of the indicated node, as well as any nodes or values contained within the selected node. Stated in terms of an XML document, the delete operation removes the entire subtree rooted at the selected node. A node that is the obligatory child of its parent node according to DTD cannot be removed. A node referenced by other nodes using IDREF cannot be removed.
- An insertion adds a node together with its descendants and values under a parent node. In an XML document, the operation inserts a subtree into a certain location. The entire subtree is given in the insertion command. The insertion of a node not conforming to the DTD or a node referencing by IDREF non-existing nodes is not allowed.
- A movement moves the node together with its descendants and values from the old position to the new position, under another node with the same type as its original parent. DTD cardinality constraints must be observed. Note that a movement does not equal deletion followed by insertion because movement preserves the identity of the node.
- A replacement can be regarded as deleting the old node and inserting the new node in one transaction. DTD and IDREF consistency need to be enforced with regard to the replacement as a whole. The deletion and insertion steps of a replacement may cause a temporary violation of constraints.

Finally, in an XML view, data from a relational tuple could appear in multiple parts of the XML document based on the view definition. We base our discussion on the assumption that when a user updates a node in the XML view, she means to update that specific part and does not expect any changes to the rest of the view. In other words, we aim to fulfill the update on the indicated piece of data while keeping the rest of the view intact. This is in contrast to the approach taken by [4], where changes to the indicated object may be cascaded to other instances in the object view. However, our algorithm framework can be easily modified to allow cascades of updates to various parts of the materialized document.

```

Metroarea (mID, mName)
State (sID, sName)
Hotel (hID, hName, starrating, pool, gym, street, city, state_id,
metro_id)
Phone (phID, phoneNo)
Confrorm (clID, croomnum, capacity, rackrate, c_h_id)
Guestroom (gID, roomnum, type, rackrate, g_h_id)
Availability (aID, startdate, enddate, price, a_r_id)
Restaurant (restID, rName, rCity)

```

(a) Relational database schema

```

<metro>
($m = SELECT mName FROM Metroarea)
<hotel>
($h = SELECT hName, starrating, pool, gym
FROM Hotel
WHERE pool > 0 AND metro_id = $m mID)
<state>
($s = SELECT sName
FROM State
WHERE sID = $h state_id)
</state>

<conference-room>
($c = SELECT croomnum, capacity
FROM Confrorm
WHERE rackrate > 2 AND c_h_id = $h hID)
<phone-number>
($p = SELECT phoneNo
FROM Phone
WHERE phID = $h hID)
</phone-number>
</conference-room>

<guest-room>
($g = SELECT roomnum, type
FROM Guestroom
WHERE rackrate > 2 AND g_h_id = $h hID)
<availability>
($a = SELECT startdate, enddate, price
FROM Availability
WHERE a_r_id = $g gID)
</availability>
</guest-room>

<nearby-restaurant>
($r = SELECT rName, rCity
FROM Restaurant
WHERE rCity = $h city)
</nearby-restaurant>
</hotel>
</metro>

```

(b) View definition

Figure 3: Example 3

2.2 Problem Definition and Assumptions

The problem we are trying to solve is defined as follows: given an underlying relational database and its schema, and an XML view definition over that database, how should the system translate an update against the XML view into corresponding updates against the underlying database without violating consistency? By *consistency*, we mean that three criteria must be satisfied. First, updates should be *side-effect free* [10]. That is, the semantics of the update performed on a materialization of the view must yield the same result as the regeneration of the XML view after performing the translated update on the

underlying database. If a side-effect free translation does not exist, the specific node is *non-updatable*. Second, the updated XML view must be consistent with the view definition and the DTD explicitly given in or derived from the view definition. Third, the updated data in the underlying database must comply with the relational schema and the constraints for the underlying database.

We base our discussion on the *reduced* system view-definition language although the algorithms we present can be modified easily for other XML view-definition languages. We divide predicates appearing in the WHERE clause into two parts. The predicates involving binding variables are called *correlation predicates*. The other predicates are called *non-correlation predicates*. The correlation predicates indicate the relationships among XML nodes. When we remove the correlation predicates, each SQL query for a single node can be regarded as a relational view that is isolated from any other nodes. We call that an *element base view*.

We make the following assumptions for the rest of the paper:

- There are no aggregates, order-by, or group operations. These operations usually make views non-updatable, as was established in prior work for relational views [10].
- The underlying relational database is in BCNF [7, 10] discuss in detail the necessity of this assumption for preservation of data dependencies.
- An element does not have more than one child node with exactly the same type and the same content.

2.3 XML View Update Example

Our running example for the remainder of this paper is drawn from a conference-planning application.

Example 3: Figure 3(a) shows the underlying database schema. Figure 3(b) defines the XML view. From the foreign-key constraints of the underlying database (which we do not show in the figure), we determine the relationship of view nodes: metro:hotel to be one-to-many (1:n), hotel:state to be many-to-one (n:1), hotel:conference-room to be 1:n, hotel:guest-room to be 1:n, guest-room:availability to be 1:n, hotel:nearby-restaurant to be many-to-many (m:n), and hotel:phone-number to be one-to-one (1:1) (phID acts both as the key and the foreign key of Phone; as a foreign key, it references to hID in the Hotel relation).

3. UPDATE TRANSLATION ALGORITHMS

3.1 Overview

As any XML update is based on a subtree instead of a single node, the particular update may affect many nodes in the subtree besides the indicated node itself. As a result, during translation, while considering the updates against the relational tuple(s) corresponding to the node itself, we also need to take into account the tuples related to the descendant (or child) nodes. This process is called *propagating the update* from the parent-node element base view to the child-node element base views.

Given an update on an XML node, we decide (1) whether the node is *updatable* for that specific update type; (2) how to *propagate* the update; and (3) what type of updates (insert,

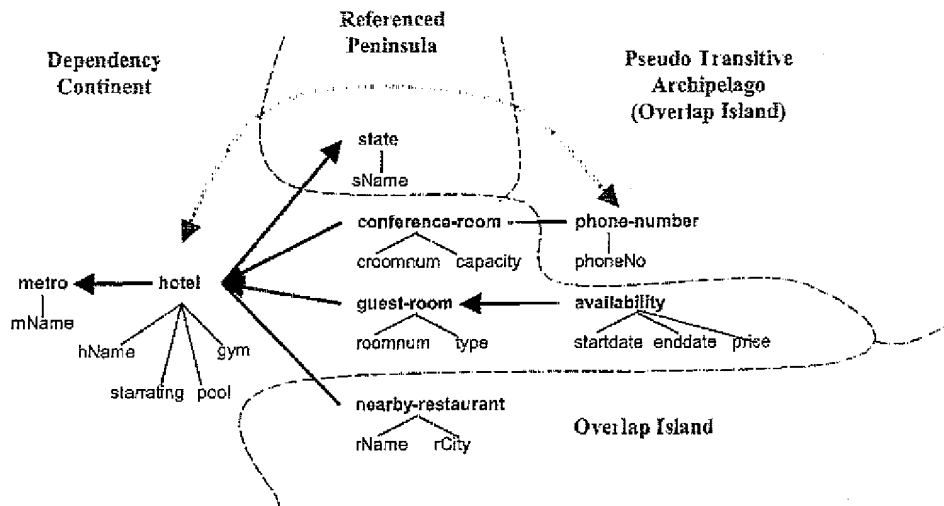


Figure 4: View-relationship graph of Example 3

delete, replace) should be performed on the element base view(s). These decisions are made by examining the *view-relationship graph* that describes the relationships between node pairs in the XML view. Based on the decisions we propose an update plan on the element base view(s). Then, we rely on a relational view-update algorithm, such as the one described in [10], to obtain the correct update plan on the underlying relational database.

In the remaining sections, we first present our algorithm to generate the view-relationship graph and the update plan for a basic case in which correlation predicates are between a parent node and a child node. Then we extend the algorithm to tackle correlation predicates between a node and its any ancestor. Finally, we discuss the changes needed to manage IDREF attributes.

3.2 XML View-relationship Graph

To visualize the relationships between node pairs in the XML view, we transform the XML view into an *XML view-relationship graph*. In an XML document, element tags and attribute names indicate the *type* of the node. A node M is called the *direct parent* of node N , if M is the parent of N in the XML view; N is called the *direct child* of M . In the view-relationship graph, we add annotated edges between each node and its direct parent to indicate the cardinality relationship. We use \leftarrow to annotate a 1:n relationship, \rightarrow to annotate a n:1 relationship, \leftrightarrow to annotate a 1:1 relationship, and finally $—$ to annotate a m:n relationship. The view-relationship graph for the running example is shown in Figure 4. The root of the view-relationship graph is the node corresponding to the root of the XML view document.

The cardinality relationship of a node pair is decided by correlation predicates in the view definition. If the correlation predicate in the child node is of the form *ForeignKey* = *\$bindingVar.Key*, where *\$bindingVar* represents the direct parent node, the relationship between the direct parent and the child is 1:n. If the correlation predicate is of the form *Key* =

\$bindingVar.ForeignKey, the relationship between the parent and the child is n:1. If the foreign key also acts as a key for the element base view, the relationship is labeled 1:1. If there is no correlation predicate between the parent and the child, or the predicate is not equality, or the comparison is not between a foreign key and its referenced key, then the relationship is labeled m:n. If there are several correlation predicates between the same pair of nodes, we follow the precedence of 1:1, n:1, 1:n, and m:n, (highest to lowest), to assign a cardinality relationship. In the above discussion, the terms *key* and *foreign key* refer to those of the element base view. For instance, in Example 2, the key of the node *conference-room* is *Confroom.cID*, and the foreign key is *Hotel.m_id*.

According to the cardinality relationship between node pairs, we partition the graph into categories. In [4], relations are grouped into categories based on object definition, including subset, ownership and reference relationships, which imply 1:1, 1:n and n:1 cardinality relationships respectively. The *many-to-many* relationship is not considered. We base our categorization entirely upon cardinality relationships indicated by foreign-key constraints in the underlying relational database. This helps us capture more semantics information. Because of the motivation provided by [4], we have chosen a convention similar to their work in naming our categories.

Definition 1 An *overlap island (OI)* is a maximal subtree of the view-relationship graph with a root N that satisfies one of the following

- 1) N has a direct parent outside the overlap island, and the relationship between N and its direct parent is m:n.
- 2) There are other nodes that get non-exclusive data from the same relation as N , and N has a relationship other than 1:n with its parent.

(Certain overlap islands will be identified in Section 3.4 as falling into the special category of transitive archipelagos.)

The root of the subtree is an **OI-root**. A node in the XML document that corresponds to a node in an overlap island is an **OI-node**. If it corresponds to an OI-root, it is an **OI-root-node**. \square

Observation 1 Given an OI-root-node its direct parent can have more than one direct child node of its type. For any OI-node N_i other nodes in the XML document may obtain their values from the same relation tuple(s) as N_i .

Definition 2 The dependency continent (DC) is a maximal subtree of the view-relationship graph such that all of the following hold

- 1) The root of the subtree is the root of the view-relationship graph
- 2) The cardinality relationship between a branch node in the subtree and its direct parent is 1:1 or n:1
- 3) No node in the subtree is a node in an overlap island.

A node in the XML document that corresponds to a node in the dependency continent is a **DC-node**. \square

Observation 2 For a given view-relationship graph, there exists only one dependency continent. Each branch node in the dependency continent has a 1:1 or n:1 relationship with its direct parent, and thus 1:1 or n:1 relationship with the root of the view-relationship graph. Given a DC-node N_i , no other node in the XML document obtains its value(s) from the same relation tuple(s) as N_i .

Proposition 1 Given a DC-node, all its ancestor nodes are also DC-nodes.

Definition 3 A referenced peninsula (RP) is a maximal subtree of the view-definition graph such that both of the following hold

- 1) The root R of the subtree has a direct parent in the dependency continent, and the relationship between R and its direct parent is 1:n.
- 2) No node in the subtree is a node in an overlap island.

The root of the subtree is called an **RP-root**. A node in the XML document that corresponds to a node in the referenced peninsula is called an **RP-node**. If it corresponds to an RP-root it is called an **RP-root-node**. \square

Observation 3 Given an RP-root-node its direct parent has only one direct child node of its type. For any RP-node N_i , other nodes in the XML document may obtain their values from the same relation tuple(s) as N_i .

Certain XML document nodes have their data in the view only once. Such nodes form the dependency continent. The other nodes may have duplications in the view. If multiple direct-parent nodes reference the same data via a foreign key, which implies the parent node can have just one child node of the given type, then the child node constitutes the root of the referenced peninsula. Else, if a direct parent can have multiple child nodes of the same type, we categorize the child nodes as being in an overlap island. The theorem below follows from this discussion.

Theorem 1 The dependency continent, referenced peninsulas, and overlap islands (including those overlap islands

characterized in Section 3.4 as transitive archipelagos) form a partition of the view-relationship graph.

In the view-relationship graph for the running example, the dependency continent includes nodes metro, hotel, conference-room, guest-room, availability, and their child leaf nodes. There is one referenced peninsula, which has the node state as its RP-root and the child leaf nodes of state. The node nearby-restaurant and phone-number, together with their child leaf nodes, form two overlap islands. We discuss the phone-number element further in Section 3.4.

The algorithm to assign categories to each XML view node is given as pseudo-code in Algorithm 1. The category assignment can be done in a single traversal of the view-relationship graph. Hence, the time complexity is $O(n)$, where n is the number of nodes in the graph.

Algorithm 1 Node Categorization

```

procedure node-cat-gen(XML.Node node)
begin
1 if (node shares underlying tables with other nodes &&
   the cardinality relationship of node and its parent is not 1:n)
2 then
3   node is in OI
4 else
5   switch (direct parent's category)
6   case DC:
7     switch (cardinality relationship of node and its parent)
8     case 1:1: node and its child leaf nodes are in DC
9     case n:1: node and its child leaf nodes are in DC
10    case 1:n: node and its child leaf nodes are in RP
11    case m:n: node and its child leaf nodes are in OI
12    end switch
13  case RP:
14    if (cardinality relationship of node and its parent is m:n)
15    then
16      node and its child leaf nodes are in OI
17    else
18      node and its child leaf nodes are in RP
19  case OI:
20    node and its child leaf nodes are in OI
21  end switch
18 for (each child branch node sub of node)
19   node-cat-gen(sub)
end

```

3.3 Update Propagation Algorithm

The updatability property and the update execution strategy of each category are different. We organize the set of possible updates by node category (DC, RP, OI) and by operation (insert, delete, move, replace). According to the update semantics described in Section 2.1, when we update a node, both the node itself and the entire subtree rooted at the node are affected. Our execution strategies follow the principles enunciated in [10]:

- 1) **No side-effects**
- 2) **One step changes:** only one step of the update execution affects a given tuple.
- 3) **Minimal changes:** no other valid strategy would require a proper subset of the database update operations.
- 4) **Simplest replacement:** no other valid strategy would make a simpler change such as a proper subset of the attributes.
- 5) **No insert-delete pairs:** replacements used instead.

The intuition for the update strategy is as follows: Updating a node may cause side-effects if and only if the underlying data to be updated appears in other parts of the view. Observation 2 indicates that a DC-node can be updated without causing side-effects. According to Observation 3, RP-root-node updates are allowed because they can be achieved by replacing the related foreign-key values for their direct parents, which are DC-nodes. Other nodes cannot be updated because of non-avoidable side-effects. To enforce foreign-key constraints for the underlying relational database, in certain cases we need to propagate deletions recursively from a parent node to its child nodes, so as to eliminate tuples containing a foreign-key value referencing the deleted tuple(s). The detailed update strategy is given as below:

- **Deletion of a branch DC-node:**

- 1) Delete the corresponding tuple in the element base view
- 2) Propagate the deletion recursively to all branch DC-children of the deleted node

- **Insertion of a branch DC-node:**

Insertion is allowed only when all of the following hold:

- 1) The OI-descendants of the inserted node, as given in the insertion, include exactly those descendant nodes that can be derived from existing tuples in the database that satisfy the correlation predicate(s)
- 2) Each branch node in the inserted subtree has a leaf child corresponding to the key of the element base view

Insertion is performed as follows:

- 1) Insert the corresponding tuple, with the foreign-key values equal to the key values of its direct parent, into the element base view
- 2) Propagate the insertion recursively to all branch DC-children of the inserted node
- 3) Propagate the insertion to its branch RP-descendants that contain new values. (Note that this is not a recursive process because according to the rules discussed below, non-root RP-nodes cannot be inserted.)

- **Movement of a branch DC-node:**

Movement is allowed only when the foreign key in the node to be moved does not itself appear in the view as a leaf node.

Movement is performed by setting the foreign-key values in the element base view of the DC-node to the key values of its new direct parent.

- **Deletion of a leaf DC-node:**

Deletion of a leaf DC-node is allowed only when the node does not correspond to a foreign key appearing in correlation predicates.

Deletion is performed by setting the corresponding attribute in the element base view to NULL.

- **Insertion of a leaf DC-node:**

Insertion is allowed only when the leaf node does not correspond to a foreign key appearing in correlation predicates.

Insertion is performed by assigning a value to the corresponding attribute in the element base view.

- **Deletion of an RP-root-node:**

Deletion of an RP-root-node is allowed only when the foreign key of the parent node does not appear in the view as a leaf node (within the parent).

Deletion is performed by setting the foreign-key values in the element base view of its direct parent to NULL.

- **Insertion of an RP-root-node:**

Insertion is allowed only when all of the following hold:

- 1) The foreign key of the parent node does not appear in the view as a leaf node (within the parent);
- 2) The OI-descendants of the inserted node, as given in the insertion, include exactly those descendant nodes that can be derived from existing tuples in the database that satisfy the correlation predicate(s)
- 3) Each branch node in the inserted subtree has a leaf child corresponding to the key of the element base view

Insertion is performed as follows:

- 1) Set the foreign-key values in the element base view of its direct parent to the key values in its element base view;
- 2) Insert the corresponding tuple into the element base view if the inserted node contains new values;
- 3) Propagate the insertion to its branch RP-descendants that contain new values.

- **Updates of a non-root RP-node: not allowed**

- **Updates of an OI-node: not allowed**

We do not enumerate the rules for replacement and movement above, with the exception of branch DC-nodes, as they can be easily derived from the rules for deletion and insertion.

Proposition 2 The update propagation algorithm correctly translates the updates on the XML view into the updates on the element base views that observe the 5 principles of [10].

Given above update strategies, it is easy to develop algorithms to generate the update plan. As an example, we give the algorithm for translating the deletion of a node from an XML view document into the element base view(s) updates. In addition to considering side-effects, we take *not-null* constraints into consideration, and explore this further in Section 4.3.

Algorithm 2 Deletion Translation

```

procedure node-delete(XMLNode node)
begin
1  switch (the category of node)
2    case DC:
3      if (node is a leaf node) then
4        if (node is not a required child of its parent) then
5          for the element base view of its parent set the
            corresponding attribute to NULL
6        else
7          node cannot be deleted according to DTD
8        else
9          delete the corresponding tuple from element base view
10         for (each child branch DC-node sub of node)
11           node-delete(sub)
12    case RP:
13      if (node is an RP-root-node) then
14        if (node is not a required child of its parent) then
15          for the element base view of its parent set the
            corresponding foreign key to NULL
16      else

```

```

17      node cannot be deleted according to DTD
18    else
19      node cannot be deleted to avoid side-effects
20  case Of:
21      node cannot be deleted to avoid side-effects
22  end switch
end

```

3.4 Extended Algorithm for Transitive Relationships

We call the relationship between a node and its direct parent a *direct relationship*, and the relationship between a node and its ancestors other than its direct parent an *indirect relationship*. In Example 3, the node phone-number has a m:n direct relationship with its parent conference-room. In addition, it has an indirect relationship with its ancestor hotel. The correlation predicate of $phID = Sh hID$ indicates that the cardinality relationship between hotel and phone-number is 1:1. This case was not covered in Section 3.2 and 3.3.

Definition 4 A transitive relationship is a non-m:n relationship defined by a correlation predicate between a node and an ancestor node other than its direct parent. The ancestor is called its transitive parent. □

Definition 5 A transitive relationship is called an effective transitive relationship if both of the following hold

- 1) The child node N has a m:n relationship with its direct parent P .
- 2) Node N has a transitive relationship with some ancestor T . T is called a real transitive parent of N . N is called a real transitive child of T , and a pseudo child of P . □

Observation 4 Given a node that has a 1:1 or 1:n relationship with its real transitive parent, its direct parent has no more than one child node of its type. Given a node that has a n:1 relationship with its real transitive parent, its direct parent can have more than one child node of its type.

In Example 3, there exists an effective transitive relationship between the node phone-number and hotel. The node phone-number is a real transitive child of the hotel node and a pseudo child of conference-room. Because the cardinality relationship between hotel and phone-number is 1:1, each conference-room node can have no more than one phone-number child.

Definition 6 A transitive relationship is called a double transitive relationship if all of the following hold

- 1) The child node N has a non-m:n relationship with its direct parent P .
- 2) N has a transitive relationship with some ancestor T .
- 3) The direct or indirect relationship between T and P is m:n. T is called the double transitive parent of N . N is called a double transitive child of T . □

Observation 5 Given a node that has a 1:1 or 1:n relationship with either its direct parent or double-transitive parent, its direct parent has no more than one child node of its type. Given a node that has n:1 relationships with both its direct parent and

its double-transitive parent, its direct parent can have more than one child node of its type.

In the view-relationship graph, we add an annotated dotted edge between a node and its transitive parent.

Definition 7 A transitive archipelago (TA) is a maximal subtree in the view-relationship graph with a root node that has a DC-node or an RP-node as its real transitive parent or double transitive parent. The root of the transitive archipelago is called a TA-root. A TA-root that has a n:1 relationship with its effective transitive DC-parent or has n:1 relationships with both its direct DC-parent and its double transitive DC-parent is a TA-DC-root. The rest TA-roots are TA-RP-roots. A node in the XML document that corresponds to a node in the transitive archipelago is a TA-node. If it corresponds to a TA-root, it is a TA-root-node. □

The nodes in the transitive archipelago are divided into transitive DC-nodes, transitive RP-nodes and transitive OI-nodes according to a categorization similar to that of Section 3.2.

Observation 6 A transitive archipelago is a subset of an overlap island.

Definition 8 A pseudo transitive archipelago (PTA) is a transitive archipelago in which the transitive parent and the direct parent of the root node have a direct or indirect relationship between them that is m:n or 1:n. The root of the subtree is called a PTA-root. A node in the XML document that corresponds to the node in the pseudo transitive archipelago is a PTA-node. If it corresponds to a PTA-root, it is a PTA-root-node. □

Observation 7 Given a PTA-root-node N , its transitive parent node P may have more than one descendant node of the same type as N 's direct parent node, and thus, may have more than one transitive child node that obtains its value from the same relation tuple as N .

Observation 8 Given a non-PTA TA-node N , its transitive parent node P has only one descendant node of the same type as N 's direct parent node, thus no other nodes in the subtree rooted at P obtain their values from the same relation tuple as N .

Notice that the update-propagation algorithm described in Section 3.3 neither allows any update to overlap islands, nor propagates any update to OI-descendant nodes. With the introduction of transitive relationship and transitive archipelagos, the special subareas of overlap islands, we need to adjust the algorithm in the following aspects:

- **Propagation to transitive children:** apply the propagations described in the algorithm to the transitive TA-children and their descendant nodes, in the same manner as for non-transitive descendant nodes in corresponding categories.
- **Updates of a non-PTA TA-node:** apply the same algorithm as that for a non-TA-node in the corresponding category.
- **Updates of a PTA-node:** not allowed.

The idea behind the above update strategy is that a TA-node that is not a PTA-node, does not share data with other nodes in the subtree rooted at its transitive parent, and thus can be treated the same as a non-TA-node in the corresponding

category. This can be inferred from Observation 8. However, according to Observation 7, a PTA-node could possibly be sharing data with other nodes in the subtree rooted at its transitive parent. Therefore to avoid side-effects, we do not allow updates on the node, but we permit propagations from the transitive parent to all such descendants.

In Example 3, the node `phone-number` belongs to a pseudo transitive archipelago, thus cannot be updated. While the updates of the node `Hotel` need to propagate to it, the updates of the node `conference room` have no effect on it.

Proposition 3 *The update propagation algorithm after the adjustments described above correctly implements the updates and observes the 5 principles of [10].*

Double transitive relationships and effective transitive relationships cover the cases where either transitive parent-direct parent relationship, or direct parent-child relationship, has min cardinality. If both are non-m:n, the problem can be transformed to the cases discussed in Sections 3.3 and 3.4, by removing the transitive parent-child relationship or the direct parent-child relationship without changing the update semantics. Furthermore, the algorithm can be easily extended to handle the case where a node has several transitive relationships with different ancestors.

3.5 Extended Algorithm with IDREF

IDREF attributes are specific to XML documents. A node referenced by other nodes using IDREF attributes is called a *referenced node*. The IDREF attribute nodes referencing it is called *referencing nodes*. We add double lines in the view-relationship graph between the referenced node and its referencing nodes, with the arrow pointing to the referenced node.

A referenced node has exactly the same updatability and update plan as common nodes in the same category, except that deletions are not allowed on a node being referenced by other nodes. Usually, referenced nodes are in the dependency island and can be safely updated.

The referencing node is an IDREF attribute node, thus a leaf node. It is treated the same as other leaf nodes in its category except that during insertion, we need to guarantee the existence of the referenced nodes.

3.6 Completeness of the Algorithm

Proposition 2 and 3 indicate the soundness of the update algorithm. Proposition 4 characterizes the scope of the algorithm.

Proposition 4 *Direct relationships, transitive relationships and IDREF reference relationships cover all explicitly given relationships in an XML view definition.*

4 SYSTEM ARCHITECTURE

To update the XML view, the system needs to parse a given update command, locate nodes for update in the XML document or directly locate related tuples in the database, translate the XML-view update into updates against underlying relations, and finally execute the updates. In addition, consistency checks need to be enforced, which may include avoiding side-effects, DTD

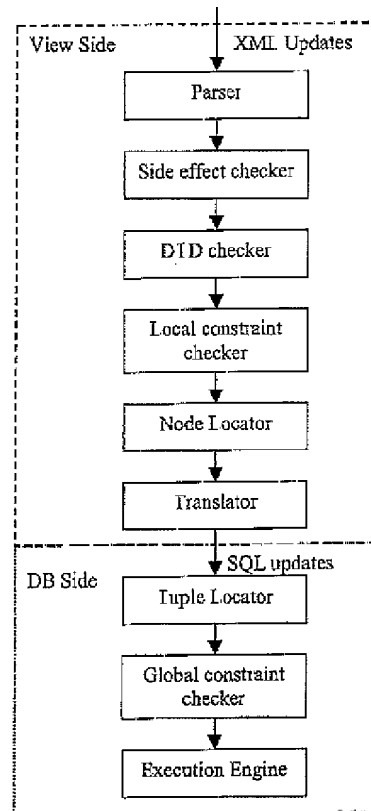


Figure 5: XML view update system architecture

validation, view-definition predicate checking, and database-constraint enforcement.

In this section, we present a system architecture that divides these tasks between the view side and the underlying database side. This division is necessary, as neither level alone can suffice. We have implemented our ideas on top of the *redacted* system. The same design idea can be applied to other XML-publishing middleware systems.

4.1 System Description

To attain higher efficiency, we complete part of the work at the view level, which means the view-level middleware system takes a first pass at the update task instead of relying on the underlying database. There are three advantages for this approach. First, doing consistency testing early and in turn finding invalid updates early can save unnecessary work in the underlying relational database. Second, static information can be collected from the view definition and the underlying relational database schema at the time of parsing the view-definition. This information, stored at the view level, can be used during the view-update process to improve performance. Third, when the work is done at the view side, we have better knowledge about the remaining update task, and therefore, can take advantage of that knowledge in optimizing the operations on the underlying database. On the other hand, the database side is more efficient

at accessing data. So any update operation or constraint checking that needs ancillary data for support is assigned to the database side. Our system architecture is shown in Figure 5.

At the view side, the parser accepts the view update command, gets information about the node to be updated, such as its tag or attribute name, the contents for insertion or replacement, the XPath and XQuery predicates for location, and so on. After that, the side-effect checker decides whether the node can be updated without causing side-effects. Then, the DTD checker determines whether the node is updatable according to the DTD, and whether the segment for insertion or replacement follows the DID³. Next, the system checks *local constraints*, including whether the inserted data violates domain requirements (declared in the underlying database schema), selection predicates (given in the view definition), etc. The three checks can be performed independently, and therefore, in parallel. After the above checking, the update command is translated into update operations against the underlying database.

The work assigned to the database side consists of three parts: locating tuples for updates, examining constraints, and executing updates. The constraints examined at this point, called *global constraints*, are across tuples and relations. Such constraints can be checked only with the knowledge of data in other tuples. Accessing data and checking those constraints can be performed more efficiently at the database side. We discuss local constraints and global constraints further in Section 4.3.

A subtle decision is when and how to locate the data for updates. One option is to compose the XQuery and XPath predicates with the view definition and transform them into an SQL update against the underlying table, leaving location task entirely to the database system. An alternative scheme is to locate the candidate nodes at the view level and update each of them. Some nodes cannot be located without processing at the view level, such as those including // or * in the XPath expression. The possibility and efficiency of pushing down the processing of location into the relational database is a subject for future research.

4.2 Information Collection while Parsing View Definition

The processing of operations at the view side requires information about the view definition and the relational schema. This information is static throughout the view-definition life, and thus can be collected while parsing the view-definition. For each node in the XML view-definition, we need the following information:

- The underlying table for the data in the node. If the node is derived from an attribute of a relation, we also record the attribute.
- The node's parent, direct children, and transitive children.
- The category of the node. (Note that these three items constitute the information built by the view-relationship graph.)

³ In case the DID is not explicitly given, we can derive it from the XML view definition and the underlying relational database schema.

- The updatability of the node according to the DID.
- The local constraints of the node.

The first three items can be obtained from the view query that defines the XML view and the algorithms defined in Section 3. Local constraints are collected from the view query and the underlying relational database schema as discussed in Section 4.3.

4.3 Constraint Satisfaction

An important task in maintaining consistency is ensuring constraint satisfaction. There are three sources for constraints: the view definition, the XML DTD, and the underlying database schema. On one hand, certain database constraints are non-enforceable at the view level either because they involve data that do not appear in the view or because the application defining the view lacks the requisite authorization. Meanwhile, certain constraints arising from the DTD, must be enforced at the view level. On the other hand, some constraints from one level can be translated into constraints at the other. In those cases, the choice of level is driven by concerns of efficiency and effectiveness.

Instead of handling the constraints where they are defined (at the sources), we categorize them into two classes based on the number of tuples used to enforce the constraints. If only one tuple is required to enforce the constraint, we call it a *local constraint* and check it at the view level; if not, we call it a *global constraint* and handle it in the relational database. In other words, some database constraints can be checked at the view level, while some view definition and DTD constraints can be translated into database constraints.

View-definition constraints come from selections that are non-correlation predicates. (We do not consider correlation and join predicates for the element base view because they are guaranteed implicitly by the update execution plan. We also ignore aggregates and order by operations, as we noted earlier.) Predicates from selections are enforced on a single tuple and therefore can be enforced at the view. In the DTD, cardinality-related constraints should be transformed and checked at the relational database, as we need data from other tuples to enforce them. Other DTD constraints can be easily enforced at the view level.

There are five types of constraints for relational database:

- 1 **Key constraints:** Key constraints are global constraints, as a base table scan or an index lookup needs to be performed to rule out the possibility of duplicate keys.
- 2 **Foreign-key constraints:** Constraints included in correlation or join predicates of the view can be enforced by update execution plans. However, if there exists a key—foreign-key relationship between a relation present in the XML view definition and a relation not involved in the view definition, we categorize the constraint as a global constraint that will be handled by the relational database.
- 3 **Domain constraints:** The effect of domain constraints is limited to the single tuple, and maybe a single attribute. They can be collected while parsing the view definition and enforced at the view.

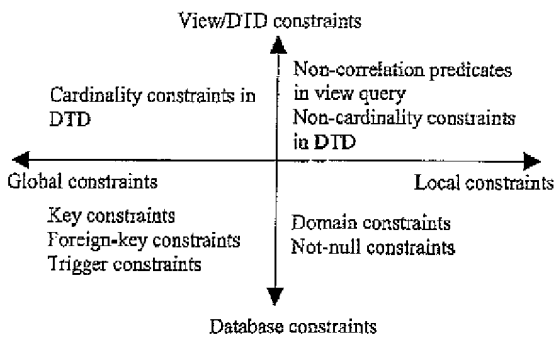


Figure 6: Summary of constraint categories

- 4 **Not-null constraints:** A not-null attribute in a relation that is used in the XML view definition should correspond to an obligatory leaf node. Such constraints are categorized as local constraints. They can be transformed into DTD constraints and enforced at the view during updates.
- 5 **Constraints defined using triggers:** These constraints are considered to be global constraints and enforced at the relational database.

In summary, local constraints, including view selection predicates, domain constraints, not-null constraints, and non-cardinality DTD constraints should be collected at the time of parsing the view definition and enforced at the view level. Others are left to the database management system. This is illustrated in Figure 6.

4.4 Implementation

We implemented the update architecture based on the *redacted* system. The architecture consists of two modules, the information collection module and the view-update execution module.

The information-collection module collects the static information described in Section 4.2 at the time when the view definition is parsed and sets up the view-relationship graph. The view-relationship graph is then translated into update plans that are persisted in the system and later used at run time.

The view-update execution module provides the interface for deletion, insertion, movement, and replacement on a given XML DOM node at run time. The execution module interacts with the relational database and the DOM interface to access the underlying data for the XML view.

The two modules are connected through the persisted update plans that provide the necessary update translation and propagation information. Experimental results show that the system operates correctly, and the performance is commensurate with direct execution without use of a view. A full performance evaluation is the subject of planned future work.

Our prototype implementation has several limitations at this time. We have not implemented an XQuery based update language, and do not use XPath-like syntax to perform node location. We believe they are not pivotal to the algorithmic ideas of this paper. Also, we have not implemented updates for views

with IDREF attributes, as view queries in the *redacted* system currently do not support IDREF(s).

5. CONCLUSION AND FUTURE WORK

In this paper, we discussed XML view updates in the context of an underlying relational database that serves not only the XML-based application, but also traditional RDBMS applications. Given a pre-existing underlying relational-database schema and an XML view defined on it, we present a framework for generating update plans to perform an update without introducing side-effects to other parts of the view.

We presented an update architecture that distributes the update subtasks between the view and underlying database, relying on the layer where efficiency is higher. Underlying relational database constraints, view constraints and DTD constraints are enforced to ensure consistency.

Although we base our discussion and implemented the update algorithms on the *redacted* system and a specific XML view-definition language, the discussions and algorithms can be easily applied to other systems and languages. Future extensions to our work could include considering element orders in XML view updates, studying the efficiency issue of locating the updated nodes at the view level and locating the related tuples at the database level, and addressing the concurrency-control issue. When completely implemented, we expect this update architecture to be the first full-featured XML view-update system for XML middleware.

6 REFERENCES

- [1] P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *Proc. of the 28th Int'l Conf. on Very Large Data Bases*, 2002.
- [2] P. Buneman, S. Khanna, and W. Ian. On propagation of deletions and annotations through views. In *Proc. of the 21st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 2002.
- [3] F. Bancilhon and N. Spyrtos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557-575, December 1981.
- [4] I. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *Proc. of the 10th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1991.
- [5] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. In *Journal of the Assoc. Comput. Mach.*, 31(4):742-760, October 1984.
- [6] Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. Technical report, Stanford University, 2001. <http://dbpubs.stanford.edu:8090/pub/2001-24>
- [7] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 3(3):381-416, September 1982.

- [8] A. Furtado and M. Casanova. Updating relational views. In W. Kim, D. S. Reinier, and D. Batory, eds., *Query processing in database systems*, Springer-Verlag Topics in Information Systems, 127-144, 1985.
- [9] M. Fernández, D. Suciu, and W. Ian. SilkRoute: Trading between relations and XML. In *Proc. WWW'9*, 2000.
- [10] A. M. Keller. Algorithms for translating view updates to database updates. In *Proc. of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1985.
- [11] P. Lehui. Design and implementation of a data manipulation processor for an XML Query Language. Technical Report, Technische Universität Darmstadt, 2001. Report KOM-D-149.
- [12] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data*, May 1979.
- [13] K. C. Sevcik and A. I. Furtado. Complete and compatible sets of update operations. In *Intl. Conf. on Management of Data (ICMOD)*, Milan, Italy, 1978.
- [14] H. Shu. Using constraint satisfaction for view update translation. In *13th European Conference on Artificial Intelligence*, 1998.
- [15] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proc. of 26th Int'l Conf. on Very Large Data Bases*. 2000.
- [16] J. Shanmugasundaram, K. Tufté, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of 25th Int'l Conf. on Very Large Data Bases*. 1999.
- [17] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. of the 20th ACM-SIGMOD Int'l Conf. on Management of Data*, 2001.